Chapter 7. Faults and Exceptions

The difference between good software and bad software is often the way in which errors and problems are handled. It's much easier to deal with processes when everything is working properly than it is to deal with failures. Structured and object-oriented programming provide lots of techniques for handling errors. Methods often return a special value, like `null` or `-1`, to indicate an error of some kind. In Java, methods frequently throw an exception indicating that something unusual has occurred. Good software is written to expect certain types of errors, and is prepared to take appropriate action. In SOAP terminology, these unusual, or exceptional, circumstances are called *faults*. Faults occur whenever a service method is not able to process input parameters and return results properly. There are endless reasons why this may occur. Common problems that result in faults are bad method parameter values, back-end problems, and improperly formatted SOAP request messages. I'm sure you can think of plenty of others, and have probably had to write code to deal with them. In this chapter we'll look at the mechanisms for generating and handling faults in SOAP.

7.1 Throwing Server-Side Exceptions in Apache SOAP

The most common way for a service to generate a fault is to throw an exception. Apache SOAP has a mechanism for handling exceptions thrown from the Java class methods that implement service methods. The information in the exception is used to generate corresponding SOAP faults to be sent back to the caller.

The contents of a SOAP fault were covered back in Chapter 2. In that section we discovered that four fault codes could be used in a fault. One of those possibilities is `SOAP-ENV:Server`, which indicates that a problem occurred while processing the message that was not related to the contents of the request. This fault code could be used if, for example, the back-end server needed to properly process the message was not available. Let's create a simple service that generates the simplest of faults so that we can see the basic mechanism used in Apache SOAP. Here's the `javasoap.book.ch7.SampleFaultService` class that implements the service `urn:SampleFaultService`:

```
package javasoap.book.ch7;
import org.apache.soap.SOAPException;
public class SampleFaultService {
   public SampleFaultService(  ) {
   }
   public int generateFault(  )
        throws Exception {
      // something bad must have happened...
      throw new Exception(  );
   }
}
```

The declaration of the `generateFault( )` method indicates that it may throw an instance of `java.lang.Exception`. In fact, that's all it does. We're simulating what would happen if the service method received a call that could not be processed because of a back-end problem.

Now let's look at a client application that calls the `generateFault` service. Up until now, we've always gotten the result from a web service by calling the `getReturnValue( )` method of the `Response` object. However, this approach assumes that the service was invoked

successfully, and doesn't handle the possibility that a fault was generated. To take faults into account, we need to ask the `Response` object if a fault was generated by calling its `faultGenerated( )` method, which returns `true` if a fault was generated and `false` otherwise. If a fault wasn't generated, we can proceed as before. If a fault was generated, we retrieve the fault from the `Response` object by calling its `getFault( )` method. This returns an instance of `org.apache.soap.Fault`, a class used to encapsulate the contents of a SOAP fault.

```
package javasoap.book.ch7;
import java.net.*;
import java.util.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;
import org.apache.soap.encoding.*;
import org.apache.soap.util.xml.*;
public class SampleClient {
   public static void main(String[] args)
      throws Exception {

      URL url = new URL(
          "http://georgetown:8080/soap/servlet/rpcrouter");

      Call call = new Call( );

      call.setTargetObjectURI("urn:SampleFaultService");
      call.setMethodName("generateFault");
      try {
         Response resp = call.invoke(url, "");
         if (resp.generatedFault( )) {
            String code = resp.getFault().getFaultCode( );
            String desc = resp.getFault().getFaultString( );
            System.out.println(code + ": " + desc);         }
         else {
            Parameter ret = resp.getReturnValue( );
            Object value = ret.getValue( );
            System.out.println(value);
         }
      }
      catch (SOAPException e) {
         System.err.println("Caught SOAPException (" +
                         e.getFaultCode( ) + "): " +
                         e.getMessage( ));
      }
   }
}
```

If you run this example you'll get the following output:

```
SOAP-ENV:Server: Exception from service object: null
```

Doing nothing more than throwing an instance of `java.lang.Exception` from our service method results in a properly formed SOAP fault with a fault code of `SOAP-ENV:Server`. The fault also contains a default value for the fault string and fault actor elements. Here's the SOAP fault returned from the `generateFault` service method:

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<SOAP-ENV:Body>
<SOAP-ENV:Fault>
<faultcode>SOAP-ENV:Server</faultcode>
<faultstring>Exception from service object: null</faultstring>
<faultactor>/soap/servlet/rpcrouter</faultactor>
</SOAP-ENV:Fault>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

But where did the fault get generated? The answer can be found in any of the deployment descriptors we've used in prior examples. Every one of those descriptors contains an `isd:faultListener` element entry that looks like this:

```
<isd:faultListener>org.apache.soap.server.DOMFaultListener
</isd:faultListener>
```

This element specifies the Java class that handles fault processing for your service. This class handles the exception that your service throws, and converts it into a SOAP fault. Unfortunately, the information that can be extracted from an instance of `java.lang.Exception` isn't all that useful. Apache SOAP includes a class called `org.apache.soap.SOAPException` that extends `java.lang.Exception` and includes methods for setting and getting the SOAP fault code and fault string. This is certainly more useful, so let's modify the service class to make use of the `SOAPException`:

```
package javasoap.book.ch7;
import org.apache.soap.SOAPException;
public class SampleFaultService {
   public SampleFaultService(  ) {
   }
   public int generateFault(  )
        throws SOAPException {
      throw new SOAPException("SOAP-ENV:Server",
                          "Data Feed Unavailable");
   }
}
```

Now the `generateFault( )` method throws an instance of `SOAPException`. Let's say that the fault occurs because the underlying data feed used by this service is not available. In this case we're required to use the `SOAP-ENV:Server` fault code, which is the first parameter of the `SOAPException` constructor. The second parameter describes the fault. These two values are used to populate the `faultcode` and `faultstring` elements of the SOAP fault. This approach also allows us to generate faults using the `SOAP-ENV:Client` fault code, which indicates that the service method encountered an improperly formatted request or some other information within the SOAP body that prevents successful processing. Think back to the stock market data feed service examples from earlier chapters. A few of those examples used attribute/value pairs as message fields. If a particular required field were missing, such as a `SYMBOL` field that provides a stock symbol, the server could generate a fault using the `SOAP-ENV:Client` fault code as follows:

```
throw new SOAPException("SOAP-ENV:Client", "Missing SYMBOL");
```

This may be all you ever need to generate adequate faults. However, in some cases, you can return much more meaningful information in SOAP faults by using the `detail` element. This element is allowed only when describing faults associated with the body of the request message. Therefore, it's not useful for describing something like an unavailable data feed, but it can certainly be useful for many other fault conditions. (Actually, I find this short-sighted because there are many situations where it would be helpful to provide details of a fault not related to the body of the request. For example, let's say the data feed is unavailable, but there is another data feed willing to accept requests. You might want the fault to include the necessary information for the client to make use of the alternative service. Of course, you could design your service to use alternative back-end resources automatically, but that's not always desirable.) In the next section, we'll look at providing fault details in the context of a `SOAP-ENV:Client` fault code.

7.2 Creating a Fault Listener in Apache SOAP

To populate a fault with a `detail` element, we need to create a new class for fault listening by implementing the `org.apache.soap.server.SOAPFaultListener` interface. This interface defines a method called `fault( )`, which is called by the Apache SOAP framework when a fault needs to be generated. The only parameter passed to this method is an instance of `org.apache.soap.server.SOAPFaultEvent`,[1] which carries an instance of `org.apache.soap.Fault` and the `SOAPException` that was thrown by the service method. The fault object contains methods for getting and setting the fault code, fault string, fault actor, and details. This is the object we want to manipulate to provide detail information.

But before we create the fault listener, we need a mechanism for providing it with the detail data. We don't have a direct path from our service methods to the fault listener, but we can take advantage of the exception mechanism. Since we've seen that the `SOAPException` that gets thrown by the service method is available to the fault listener, we can inherit from `SOAPException` and create a new class that meets our needs. But before we do that, we need a class to carry our detail information. Let's use a `java.util.Hashtable`. That way we can set up any number of attribute/value pairs to represent the fault detail information. With all this in mind, let's create a new exception type with a property called `Detail` that's an instance of `Hashtable`.

```
package javasoap.book.ch7;
import org.apache.soap.*;
import java.util.Hashtable;
public class FeedException extends SOAPException {
   Hashtable _detail;
   public FeedException(String fcode, String fstring) {
      super(fcode, fstring);
   }
   public void setDetail(Hashtable detail) {
      _detail = detail;
   }
   public Hashtable getDetail(  ) {
      return _detail;
   }
}
```

[1] If you're familiar with the JavaBeans patterns for events you should recognize this naming convention. The event that has taken place is a `SOAPFault`. The listener interface is named by appending `Listener` to the event name, and the object carrying the event information is named by appending `Event`.

The code is pretty simple. Since `FeedException` extends `SOAPException`, we can use an instance of this new class in place of the `SOAPException` instance we used earlier. The constructor takes the fault code and fault string as parameters and passes them along to the superclass constructor. We have set and get accessors for the `Detail` property, defined as an instance of `Hashtable`.

Now we can modify the service class to throw an instance of `FeedException` and populate it with some detail information. Here's the new `SampleFaultService` code with the changes. We've included `reasonCode`, `reasonDescription`, `alternateProvider`, and `alternateContact` fields in the details with appropriate values. Those details, contained in a hash table, are passed to the `FeedException` instance using the `setDetail( )` method. We can throw the `FeedException` successfully because it's a subclass of `SOAPException`. Alternately, we could modify the method definition to specify that it throws a `FeedException`.

```
package javasoap.book.ch7;
import org.apache.soap.SOAPException;
import java.util.Hashtable;
public class SampleFaultService {
   public SampleFaultService(  ) {
   }
   public int generateFault(  )
        throws SOAPException {
      FeedException e = new FeedException("SOAP-ENV:Server",
                            "Data Feed Unavailable");
      Hashtable detail = new Hashtable(  );
      detail.put("reasonCode", "199");
      detail.put("reasonDescription", "Power Outage");
      detail.put("alternateProvider", "www.mindstrm.com");
      detail.put("alternateContact", "rob@mindstrm.com");
      e.setDetail(detail);
      throw e;
   }
}
```

Now we can work on the new fault listener class. Let's call it `javasoap.book.ch7.FeedFaultListener`. It implements the `org.apache.soap.server.SOAPFaultListener` interface and implements the required `fault( )` method.

```
package javasoap.book.ch7;
import org.apache.soap.*;
import org.apache.soap.server.*;
import java.util.*;
import org.w3c.dom.*;
import org.apache.soap.util.xml.*;
import javax.xml.parsers.DocumentBuilder;

public class FeedFaultListener
        implements SOAPFaultListener {
   public FeedFaultListener(  ) {
   }
   public void fault(SOAPFaultEvent event) {

      FeedException ex = (FeedException)event.getSOAPException(  );
      Vector v = new Vector(  );
```

```
        DocumentBuilder builder =
                 XMLParserUtils.getXMLDocBuilder( );
        Document doc = builder.newDocument( );
        Hashtable detail = ex.getDetail( );
        Enumeration e = detail.keys( );
        while (e.hasMoreElements( )) {
           String name = (String)e.nextElement( );
           String value = (String)detail.get(name);
           Element elem = doc.createElement(name);
           Text txt = doc.createTextNode("");
           txt.setData(value);
           elem.appendChild(txt);
           v.addElement(elem);
        }

        event.getFault( ).setDetailEntries(v);
    }
}
```

Before describing the logic in detail, let's look at a high level at what the `fault( )` method has to do. It receives a `SOAPFaultEvent` in response to the `FeedException` thrown by the service method. This event includes references to both the `FeedException` and the actual `Fault` that will be sent to the caller. So we need to extract the detailed information we stuck into the `FeedException`, turn it into a group of XML elements that can be added to the `Fault`, and then stick these entries into the `Fault` so that they'll be in the SOAP document sent back to the caller.

The `fault( )` method starts by getting the `FeedException` that was thrown by the service method. To get the exception, we call the `getSOAPException( )` method on the `event`. The result is cast to a `FeedException`, since that's where we'll find the details of the fault. An instance of `java.util.Vector` is created to store the elements of the detail section as we create them.

Now we need to create XML child elements of the `detail` element, making use of APIs for manipulating XML that are in no way specific to SOAP. If you want to learn about manipulating XML in Java, see Brett McLaughlin's book, *Java and XML* (O'Reilly). We're not doing anything complicated, though, so you should be able to follow along without much trouble. We call `XMLParserUtils.getXMLDocBuilder` to get an instance of a `DocumentBuilder` interface, which in turn is used to create a new document. This `Document` is an XML document that we'll use to populate our detail entries.

The details are retrieved using the `getDetail( )` method of the `FeedException`. The next step is to enumerate the hash table entries, pulling out the name/value pairs that constitute the detail data. For each pair, we create a new XML element by calling `doc.createElement( )` using the `name` as the element name. We then create a text node and set its value using the `value` retrieved from the hash table. Now the text node is appended to the element, and the element is added to the vector we created earlier. When we've done this for all of the pairs, the vector of detail elements is added to the `Fault` object by passing it as a parameter to its `setDetailEntries( )` method.

We now need to redeploy the service, this time using the `FeedFaultListener` class as the value of the `isd:FaultListener` element in the deployment descriptor:

```
<isd:faultListener>
javasoap.book.ch7.FeedFaultListener
</isd:faultListener>
```

If you run the client application, the fault returned from the server will now include a detail section. The client doesn't look for a detail section, but we can see the result of our work by looking at the SOAP envelope returned from the server:

```
<SOAP-ENV:Envelope
 xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<SOAP-ENV:Body>
 <SOAP-ENV:Fault>
  <faultcode>SOAP-ENV:Server</faultcode>
  <faultstring>Data Feed Unavailable</faultstring>
  <faultactor>/soap/servlet/rpcrouter</faultactor>
  <detail>
    <alternateProvider>www.mindstrm.com</alternateProvider>
    <reasonDescription>Power Outage</reasonDescription>
    <alternateContact>rob@mindstrm.com</alternateContact>
    <reasonCode>199</reasonCode>
  </detail>
 </SOAP-ENV:Fault>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The four detail fields are included as children of the `detail` element. For this information to be useful, we want to get at the details from our client application, so let's do that now. Here is a modified version of the client application that extracts the details we sent with the fault:

```
package javasoap.book.ch7;
import java.net.*;
import java.util.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;
import org.apache.soap.encoding.*;
import org.apache.soap.util.xml.*;
import org.w3c.dom.*;
public class SampleClient {
   public static void main(String[] args)
      throws Exception {

      URL url = new URL(
          "http://georgetown:8080/soap/servlet/rpcrouter");

      Call call = new Call(  );

      call.setTargetObjectURI("urn:SampleFaultService");
      call.setMethodName("generateFault");
      try {
         Response resp = call.invoke(url, "");
         if (resp.generatedFault(  )) {
            Fault fault = resp.getFault(  );
            String code = fault.getFaultCode(  );
            String desc = fault.getFaultString(  );
            System.out.println(code + ": " + desc);
            Vector v = fault.getDetailEntries(  );
            int cnt = v.size(  );
```

```
            for (int i = 0; i < cnt; i++) {
               Element n = (Element)v.elementAt(i);
               Node nd = n.getFirstChild( );
               System.out.println(n.getNodeName( ) + ": " +
                                  nd.getNodeValue( ));
            }
         }
         else {
            Parameter ret = resp.getReturnValue( );
            Object value = ret.getValue( );
            System.out.println(value);
         }
      }
      catch (SOAPException e) {
         System.err.println("Caught SOAPException (" +
                        e.getFaultCode( ) + "): " +
                        e.getMessage( ));
      }
   }
}
```

If `resp.generatedFault( )` returns `true`, we print out the fault code and fault string values just as we did before. Next, we call `fault.getDetailEntries( )`, which returns a vector containing the detail elements. We simply iterate over the vector, casting each object retrieved to an instance of `org.w3c.dom.Element` and calling the `getFirstChild( )` method on the resulting element. Now we display the element name and the node data. If you run the example you should see the following output:

```
SOAP-ENV:Server: Data Feed Unavailable
alternateProvider: www.mindstrm.com
reasonDescription: Power Outage
alternateContact: rob@mindstrm.com
reasonCode: 199
```

7.3 Throwing and Catching Exceptions in GLUE

The concepts covered for faults and exceptions in Apache SOAP are similar to those used in GLUE, although some of the techniques are different. Actually, GLUE has a few ways of dealing with exceptions, so if you're interested in all the possibilities, I encourage you to read the GLUE user's guide; we're going to cover only one technique here.

Let's skip the basics and get right into an example that returns faults with a detail section. GLUE includes its own SOAP exception class called `electric.net.soap.SOAPException`. It's not the same as the Apache `SOAPException`, so don't confuse the two. GLUE's `SOAPException` allows you to set the fault code, fault string, fault actor, and details, and has several constructors using a variety of combinations of these fault element values. Therefore, you don't have to write a fault handler to get the detail information into the fault; just throw an instance of `SOAPException` with the appropriate values. Let's create a service called `urn:AnotherFaultService` and implement it with the `AnotherFaultService` class:

```
package javasoap.book.ch7;
import electric.net.soap.SOAPException;
import electric.xml.*;
public class AnotherFaultService {
   public int generateFault(  )
        throws SOAPException {
      Element elem = new Element("detail");
      Element sub = elem.addElement("reasonCode");
      sub.addText("199");
      sub = elem.addElement("reasonDescription");
      sub.addText("Power Outage");
      sub = elem.addElement("alternateProvider");
      sub.addText("www.mindstrm.com");
      sub = elem.addElement("alternateContact");
      sub.addText("rob@mindstrm.com");
      SOAPException e = new SOAPException(
                           "Data Feed Unavailable",
                           SOAPException.SERVER,
                           "urn:AnotherFaultService",
                           elem);
      throw e;
   }
}
```

In this class, we generate the detail section before creating an instance of `SOAPException`. GLUE includes a simple XML API, which I've used here to create the `detail` element. The first step is to create an instance of `electric.xml.Element` with a tag name of `"detail"`. Following that, I add new elements to the `detail` element by calling `elem.addElement( )`, with the tag name of the added element as the parameter. The data is set by calling the `addText( )` method of the element returned from `elem.addElement( )`. Once we have the `detail` element, we're ready to create the `SOAPException`. The parameters to the constructor are the fault string, the fault code, the fault actor, and the XML element containing the details. Here's the code used to deploy `urn:AnotherFaultService`:

```
package javasoap.book.ch7;
import electric.util.Context;
import electric.server.http.HTTP;
import electric.registry.Registry;
public class FaultServiceApp {

   public static void main( String[] args )
                throws Exception {
      String ns = "urn:AnotherFaultService";
      HTTP.startup("http://georgetown:8004/glue");
      Context context = new Context(  );
      context.addProperty("activation", "application");
      context.addProperty("namespace", ns);
      Registry.publish(ns,
         javasoap.book.ch7.AnotherFaultService.class, context );
   }
}
```

The Java interface for binding to this service contains the `generateFault( )` method, which is declared to throw an instance of `electric.net.soap.SOAPException`:

```
package javasoap.book.ch7;
import electric.net.soap.SOAPException;
public interface IAnotherFaultService {
  int generateFault(  ) throws SOAPException;
}
```

GLUE will handle the details of putting the fault on the wire, as well as turning that fault back into an instance of `electric.net.soap.SOAPException` on the client side. Here's the client application that invokes the `generateFault( )` method of `urn:AnotherFaultService`:

```
package javasoap.book.ch7;
import electric.registry.RegistryException;
import electric.registry.Registry;
import electric.net.soap.SOAPException;
import electric.xml.*;
public class FaultServiceClient {
   public static void main(String[] args) throws Exception
   {
      try {
         IAnotherFaultService srv =
           (IAnotherFaultService)Registry.bind(
             "http://georgetown:8004/glue/urn:AnotherFaultService.wsdl",
             IAnotherFaultService.class);
         srv.generateFault(  );

      }
      catch (SOAPException se) {
         System.out.println(se.getSOAPCode(  ));
         System.out.println(se.getMessage(  ));
         System.out.println(se.getSOAPActor(  ));
         System.out.println(se.getSOAPDetailElement(  ));
      }
      catch (RegistryException e) {
         System.out.println(e);
      }
  }
}
```

We print the contents of the fault in the `catch` block for `SOAPException`. The fault code, fault string, and fault actor are retrieved using the `SOAPException` method calls `getSOAPCode( )`, `getMessage( )`, and `getSOAPActor( )`, and the details can be retrieved as an XML element by calling `getSOAPDetailElement( )` . If we wanted to process the fields of the detail section, we could look at the subelements and their values individually. In this case, we're just displaying its contents by taking advantage of the `toString( )` method of the detail `Element`. If you run this example you'll get the following output:

```
Server
Data Feed Unavailable
urn:AnotherFaultService
<detail>
  <reasonCode>199</reasonCode>
  <reasonDescription>Power Outage</reasonDescription>
  <alternateProvider>www.mindstrm.com</alternateProvider>
  <alternateContact>rob@mindstrm.com</alternateContact>
</detail>
```

The only thing that may look unusual is that the fault code, displayed on the first line, does not include its namespace qualifier; the `SOAPException` implementation apparently strips it off. That's probably a good idea, since the qualifier is meaningless unless you know its value as well. You can see what I mean by looking at the SOAP envelope returned by the server:

```
<soap:Envelope
  xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
  xmlns:xsd='http://www.w3.org/2001/XMLSchema'
  xmlns:soap='http://schemas.xmlsoap.org/soap/envelope/'
  xmlns:soapenc='http://schemas.xmlsoap.org/soap/encoding/'
  soap:encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'>
    <soap:Body>
      <soap:Fault>
        <faultcode>soap:Server</faultcode>
        <faultstring>Data Feed Unavailable</faultstring>
        <faultactor>urn:AnotherFaultService</faultactor>
        <detail>
          <reasonCode>199</reasonCode>
          <reasonDescription>Power Outage</reasonDescription>
          <alternateProvider>www.mindstrm.com</alternateProvider>
          <alternateContact>rob@mindstrm.com</alternateContact>
        </detail>
      </soap:Fault>
    </soap:Body>
  </soap:Envelope>
```

Generating and handling SOAP faults is an integral part of writing applications using SOAP. You can't ignore it any more than you can ignore exceptions and exception handling in Java. Luckily, the techniques for working with faults are straightforward, and with a little extra work an endless amount of detail can be included.

The best software always expects faults to take place. The trick is to design for them ahead of time, and not be surprised by faults that you didn't plan for. That's not SOAP; that's software design. Get your fault processing design right, and you'll have no trouble implementing it using SOAP.